

F/G 9/2

F30602-77-C-0194

RAOC-TR-80-6-VOL-1

NL

106
A22332

10

END
DATE
FILMED
5-80
DTIC

(12) LEVEL II
NEW

RADC-TR-80-6, Vol I (of two)
Interim Report
February 1980



ADA 083512

A MATCHED PROJECT EVALUATION OF MODERN PROGRAMMING PRACTICES Management Report on the ASTROS Plan

General Electric Company

Phil Milliman
Bill Curtis

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

DTIC
ELECTE
S D
APR 28 1980
B

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

DOC FILE COPY

80 4 25 016

This report has been reviewed by the RADC Public Affairs Office and is releasable to the National Technical Information Office (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

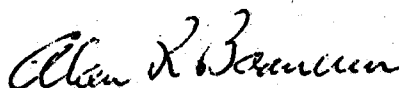
RADC-TR-80-6, Vol I (of two) has been reviewed and is approved for publication.

APPROVED:



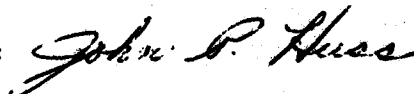
DONALD F. ROBERTS
Project Engineer

APPROVED:



ALAN R. BARNUM, Asst Chief
Information Sciences Division

FOR THE COMMANDER:



JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC TR-80-6 Vol 1 (of two)	2. GOVT ACCESSION NO. AD-A083512	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A MATCHED PROJECT EVALUATION OF MODERN PROGRAMMING PRACTICES. Volume 1. Management Report on the ASTROS Plan.		5. TYPE OF REPORT & PERIOD COVERED Interim Report. 1 Sep 77 - 30 Nov 78
6. AUTHOR(s) Phil/Milliman Bill/Curtis		7. PERFORMING ORG. REPORT NUMBER 791SP0051
8. CONTRACT OR GRANT NUMBER(s) F30602-77-C-0194 new		9. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63728F 25280103 14 41
10. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441		11. REPORT DATE Feb 1980
12. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same		13. NUMBER OF PAGES 53
14. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		15. SECURITY CLASS. (of this report) UNCLASSIFIED
16. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same		17. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A
18. SUPPLEMENTARY NOTES RADC Project Engineer: Donald E. Roberts (ISIS)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Modern Programming Practices Software Data Base Chief Programmer Teams Matched Project Evaluation Software Science Structured Programming Software Error Data Software Management		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Rome Air Development Center (RADC) and the Space and Missile Test Center (SAM-TEC) at Vandenberg Air Force Base have jointly developed a plan for improving software development called Advanced Systematic Techniques for Reliable Operational Software (ASTROS). This system provides guidelines for applying the following modern programming practices to software development: → next page (Cont'd)		

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

407-113

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

→
• Structured design and testing
• HIPO charts,
• Chief programmer teams,
• Structured coding,
• Structured walk-throughs, and
• Program support library.

In order to test the utility of these techniques, two development projects (non-real-time) sponsored by SAMTEC from the Metric Integrated Processing System were chosen for a quasi-experimental comparison. This system provides control and data analysis for missile launches. The Launch Support Data Base (LSDB) was developed under the guidelines of the ASTROS plan, while the Data Analysis Processor (DAP) was developed using conventional techniques.

This report is the first of two volumes describing the performance of the LSDB project implemented under ASTROS. This volume presents a condensed report of the results of this study which has been prepared for a non-scientific and/or managerial audience. Readers wanting a detailed account of the theories and analyses underlying these results are referred to Volume II. Where material has been substantially condensed in this volume, a reference will be provided to the corresponding section in Volume II where further elaboration of the material is available.

→ The performance of the LSDB project was comparable to that of similar sized software development projects on numerous criteria. The amount of code produced per man-month was typical of conventional development efforts. Nevertheless, the performance of the LSDB project was superior to that of the DAP project. Thus, the benefits of the modern programming practices employed on the LSDB project were limited by the constraints of environmental factors such as computer access and turnaround time. ←

While the results of this study demonstrated reduced programming effort and improved software quality for a project guided by modern programming practices, no causal interpretation can be reliably made. That is, with only two projects and no experimental controls, causal factors can only be suggested, not isolated. The generalizability of these results to other projects is uncertain. The data also do not allow analyses of the relative values of each separate practice. Nevertheless, the results of this study suggest that future evaluations will yield positive results if constraints in the development environment are properly controlled.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

Table of Contents

<u>Title</u>	<u>Page</u>
Abstract	i
Table of Contents	iii
List of tables	v
List of figures	vi
 1. Introduction	 1
1.1 Purpose for this Research	1
1.2 Advanced Systematic Techniques for Reliable Operational Software (ASTROS)	 2
1.2.1 Structured design and testing	2
1.2.2 HIPO charts	3
1.2.3 Chief programmer teams	3
1.2.4 Structured coding	5
1.2.5 Structured walk-throughs	5
1.2.6 Program support library	7
1.3 Metric Integrated Processing System (MIPS)	7
1.3.1 Launch Support Data Base (LSDB)	8
1.3.2 Data Analysis Processor (DAP)	8
1.4 Project Environment	9
 2. Findings	 11
2.1 Comparative Analyses: LSDB Versus DAP	12
2.1.1 Descriptive data	12
2.1.2 Level of Technology: Putnam's Model	16
2.1.2.1 Theory	16
2.1.2.2 Results	17
2.1.3 Programming Scope and Time: Halstead's Model	17
2.1.3.1 Theory	17
2.1.3.2 Results	18
2.1.4 Complexity of Control Flow: McCabe's Model	20
2.1.4.1 Theory	20
2.1.4.1 Results	20
2.1.5 Software Quality Metrics	22
2.1.6 Comparison to RADC Database	24
2.2 Error Analyses	24
2.2.1 Error Categories	24
2.2.1.1 Descriptive data	24
2.2.1.2 Comparison with TRW data	27
2.2.2 Error Trends over Time	27
2.2.3 Post-Development Errors	29
 3. Interviews	 33
3.1 Chief Programmer Teams	33
3.2 Design and Coding Practices	36

<u>Title</u>	<u>Page</u>
4. Conclusions	38
4.1 Current Results	38
4.2 Future Approaches to Evaluative Research	40
5. References	42

List of Tables

<u>Table</u>	<u>Title</u>	<u>Page</u>
1	Comparison of Descriptive Data from LSDB and DAP	13
2	Comparison Between LSDB and DAP on Halstead's Metrics . .	19
3	McCabe's v(G) and Software Quality Metrics for Selected Subroutines from LSDB and DAP	21
4	Frequencies of Error Categories	26

DTIC
ELECTE
S **D**
 APR 28 1980
B

ACCESSION for		
NTIS	White Section	<input checked="" type="checkbox"/>
DDC	Buff Section	<input type="checkbox"/>
UNANNOUNCED		<input type="checkbox"/>
JUSTIFICATION _____		
BY _____		
DISTRIBUTION/AVAILABILITY CODES		
Dist.	A/AIL.	and/or SPECIAL
A		

List of Figures

<u>Figure</u>	<u>Title</u>	<u>Page</u>
1	HIPO chart	4
2	Control structures allowed in structured programming . . .	6
3	Chronological manpower loadings by project	14
4	Percentage of total effort expended in each phase of development	15
5	Decomposition to essential complexity	23
6	Scatterplot for the regression of delivered source lines of code on productivity	26
7	Comparison of error distributions between LSDB and three TRW studies.	28
8	Error ratio by month	30
9	Prediction of post-development errors	31

1. INTRODUCTION

1.1 Purpose for This Research

In 1973 Boehm chronicled the tremendous impact of software on the cost and reliability of advanced information processing systems. Recently, DeRoze and Nyman (1978) estimated the yearly cost for software within the Department of Defense to be as large as three billion dollars. De Roze (1977) reported that perhaps 115 major defense systems depend on software for successful operation. Nevertheless, the production and maintenance of software for Defense is frequently inefficient.

In an effort to improve both software quality and the efficiency of software development and maintenance, a number of techniques have been developed as alternatives to conventional programming practices. These modern programming practices include such techniques as structured coding, structured design, program support libraries, and chief programmer teams (W. Myers, 1978). The New York Times project implemented by IBM (Baker, 1972) was lauded as a successful initial demonstration of these techniques. Yet, some problems appear to have resulted from an early release of the system (Yourdon Report, 1976). Considerable variability has been reported in subsequent studies on the effect of these techniques for various project outcomes (Belford, Donahoo, & Heard, 1977; Black, 1977; Brown, 1977; Walston & Felix, 1977). Many evaluations have rested solely on subjective opinions obtained in questionnaires. There is a critical need for empirical research evaluating the effects of modern programming practices on software development projects.

Rome Air Development Center (RADC) and the Space and Missile Test Center (SAMTEC) at Vandenberg Air Force Base have jointly developed a plan (Lyons & Hall, 1976) for improving software development called Advanced Systematic Techniques for Reliable Operational Software (ASTROS). This plan describes several modern programming practices which should result in more reliable and less expensively produced and maintained software. In order to evaluate the utility of these techniques, RADC is sponsoring research into their use on a software development project at SAMTEC.

Two development projects from the non-real-time segment of the Metric Integrated Processing System (MIPS) were chosen for comparison. The MIPS system provides control and data analysis in preparation for missile launches. The Launch Support Data Base (LSDB) segment of MIPS was implemented

under the ASTROS plan while the Data Analysis Processor (DAP) was implemented with conventional software development techniques.

Data from LSBD were compared with results from DAP, allowing a quasi-experimental comparison of two similar projects in the same environment, one implemented under the ASTROS plan. Data from these two projects were also compared with the results from software projects elsewhere in industry. Because there is little control over many of the influences on the two projects, a causal relationship between programming practices and performance cannot be proved, but the data can be investigated for evidence of their effects.

1.2 Advanced Systematic Techniques for Reliable Operational Software (ASTROS)

The ASTROS plan was a joint effort by SAMTEC and RADC to implement and evaluate modern programming practices in an Air Force operational environment. ASTROS applied these practices to selected programming projects in order to demonstrate empirically the premise that these techniques would yield lower costs per line of code, more reliable software, more easily maintained code, and less schedule slippage.

The ASTROS project focused on three objectives: 1) an investigation and validation of structured programming tools and concepts, 2) improving management aspects of structured programming, and 3) empirical measurement of project process and outcomes. The core of the ASTROS plan was the specification of a set of modern programming practices. The implementation of these practices by the LSDB project team as described by Salazar and Hall (1977) is discussed below.

1.2.1 Structured design and testing - is the practice of top-down development or stepwise refinement (Stevens, Myers, & Constantine, 1974; Yourdon & Constantine, 1975). Each subsystem is designed from the control sections down to the lowest level subroutines prior to the start of coding. Thus, the highest level units of a system or subsystem are coded and tested first. Top-down implementation does not imply that all system components at each level must be finished before the next level is begun, but rather the fathers of a unit must be completed before this unit can be coded.

Since higher level units will normally invoke lower level units, dummy code must be substituted temporarily for these latter units. The required dummy units (program stubs) may be generalized, prestored on disk, and included

automatically by the linkage editor during a test run, as in the case of a CALL sequence. Although program stubs normally perform no meaningful computations, they can output a message for debugging purposes each time they are executed. Thus, it is possible to exercise and check the processing paths in the highest level unit before initiating implementation of the lower level units it invokes. This procedure is repeated, substituting actual program units for the dummy units at successively lower levels until the entire system has been integrated and tested. Program units at each level are fully integrated and tested before coding begins at the next lower level.

1.2.2 HIPO charts - Hierarchical Input-Process-Output charts are diagrammatic representations of the operations performed on the data by each major unit of code (Katzen, 1976; Stay, 1974). A HIPO chart is essentially a block diagram showing the inputs into a functional unit, the processes performed on that data within the unit, and the output from the unit (Figure 1). There was one HIPO per functional unit, with the processing in one unit being expanded to new HIPOs until the lowest level of detail was reached. The hierarchical relationships among the HIPO charts are displayed in a Visual Table of Contents.

1.2.3 Chief programmer teams - are organized so that functional responsibilities such as data definition, program design, and clerical operations are assigned to different members (Baker, 1972; Baker & Mills, 1973). This approach results in better integration of the team's work, avoiding the isolation of individual programmers that has often characterized programming projects. The chief programmer team is made up of 3 core members and optional support members who are programmers.

- Chief programmer - is responsible to the project manager for developing the system and managing the programming team. He carries technical responsibility for the project including production of the critical core of the programming system in detailed code, direct specification of all other codes required for system implementation, and review of the code integration.
- Back-up programmer - supports the chief programmer at a detailed task level so that he can assume the chief programmer's role temporarily or permanently if required. In the LSDB project, he was responsible for generating 80% of the code.
- Librarian - assembles, compiles, and link-edits

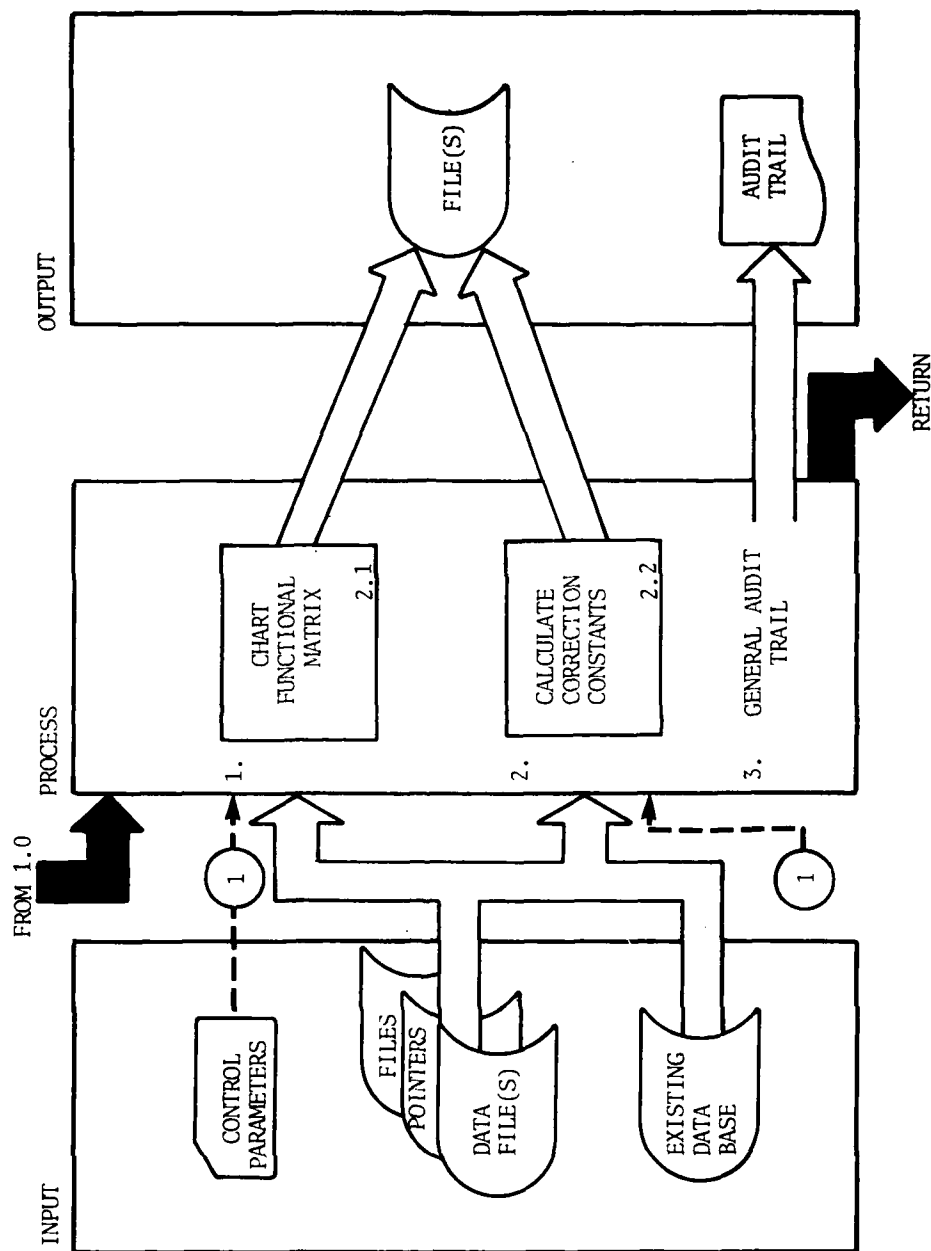


Figure 1. HIPO Chart

the programs submitted by project programmers. The librarian is responsible for maintaining the library, including work books, record books, subroutines and functions, and configuration control for all source code not maintained by the program support library.

There was a tester assigned to the LSDB project who was not formally a member of the chief programmer team. Nevertheless, he attended all walk-throughs and knew the system almost as well as the project programmers. His primary responsibility was to test the generated code for accuracy and proper function. When a problem arose, he would send an 'Action Item' to the chief programmer regarding the test results.

1.2.4 Structured coding - is based on the mathematically proven Structure Theorem (Mills, 1975) which holds that any proper program (a program with one entry and one exit) is equivalent to a program that contains as control structures only:

- Sequence - two or more operations in direct succession.
- Selection - a conditional branching of control flow. Selection control structures are:
 1. IF-THEN-ELSE
 2. CASE
- Repetition - a conditional repetition of operations while a condition is true, or until a condition becomes true. Repetition control structures are:
 1. DO WHILE
 2. REPEAT UNTIL

These control structures are illustrated in Figure 2. The implementation of these constructs into a computer language allows the implementation of a simpler, more visible control flow which results in more easily understood programs (Dijkstra, 1972).

1.2.5 Structured walk-throughs - A structured walk-through is a review of a developer's work (program design, code, documentation, etc.) by fellow project members invited by the developer. Not only can these reviews locate errors earlier in the development cycle, but reviewers are exposed to other design and coding strategies. A typical walk-through is scheduled for one or two hours. If the objectives

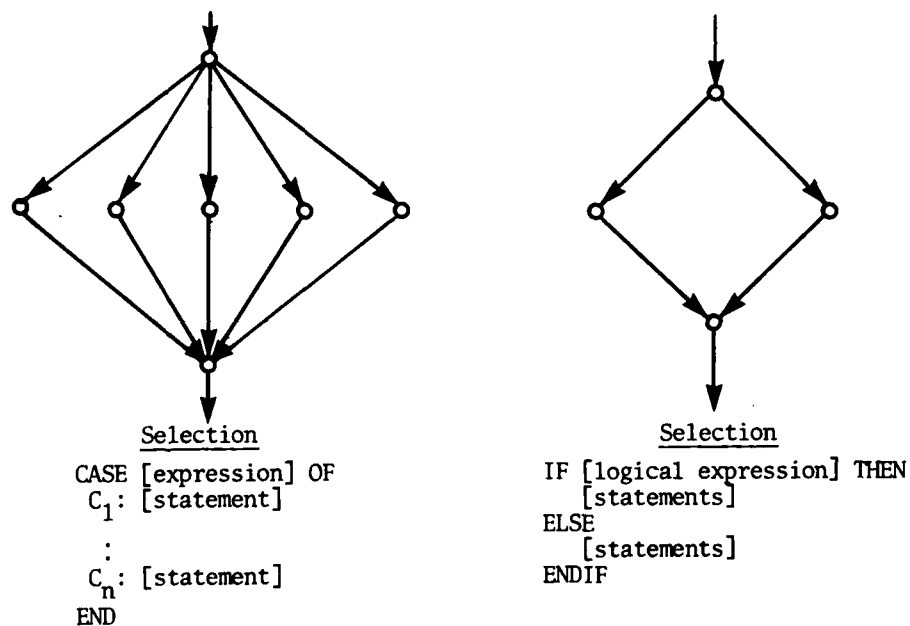
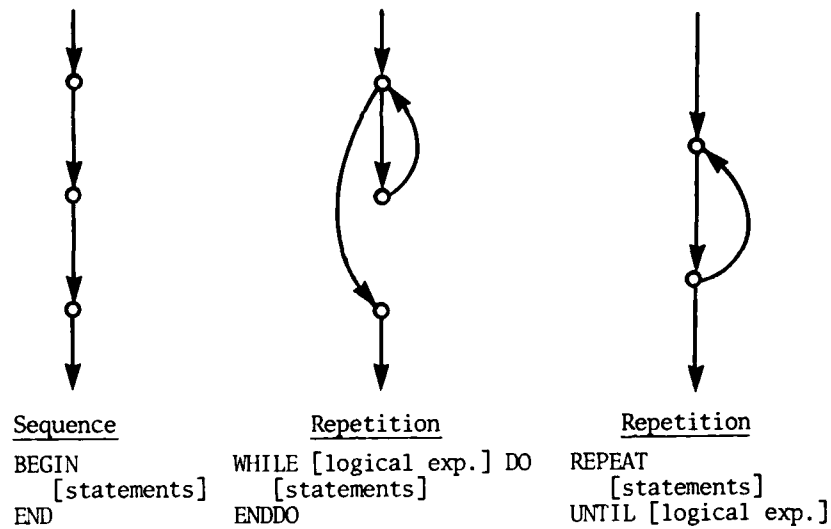


Figure 2. Control structures allowed in structured programming

have not been met by the end of the session, another walk-through is scheduled.

During a walk-through reviewers are requested to comment on the completeness, accuracy, and general quality of the work presented. Major concerns are expressed and identified as areas for potential follow-up. The developer then gives a brief tutorial overview of his work. He next walks the reviewers through his work step-by-step, simulating the function under investigation. He attempts to take the reviewers through the material in enough detail to satisfy the major concerns expressed earlier in the meeting, although new concerns may arise.

Immediately after the meeting, the appointed moderator distributes copies of the action list to all attendees. It is the responsibility of the developer to ensure that the points of concern on the action list are successfully resolved and reviewers are notified of the actions taken. It is important that walk-through criticism focus on error detection rather than fault finding in order to promote a readiness to allow public analysis of a programmer's work.

1.2.6 Program support library - The Applied Data Research (ADR) LIBRARIAN software was chosen as the program support library for the LSDB project. A major reason for this choice was its versatility of tools for the IBM 360/65 system. A program support library provides a vehicle for the organization and control of a programming project, the communications among development personnel, and the interface between programming personnel and the computer system. When used with top-down structured programming, the program support library maintains a repository of data necessary for the orderly development of computer programs.

The ADR LIBRARIAN generates a weekly subroutine report which indicates the subroutine size, number of updates, number of runs, etc., during the preceding report period. It is a source program retrieval and maintenance system designed to eliminate problems involved in writing, maintaining, testing, and documenting computer programs. Source programs, test data, job control statements, and any other information normally stored on cards was stored by the ADR LIBRARIAN on tape or disk and updated by a set of simple commands. Data is also stored in hardcopy form in project notebooks. The ADR LIBRARIAN includes the necessary computer and office procedures for controlling and manipulating this data.

1.3 Metric Integrated Processing System (MIPS)

MIPS was designed to support SAMTEC operations as the

primary source of metric (positional) data processing associated with missile, aircraft, satellite testing, or trajectory measurement activities. The critical function of MIPS is to determine range safety information before launch and establish controls for any potential flight path or abort problems. The MIPS development effort consisted of reprogramming and integrating several programs which performed similar functions less flexibly and used the same hardware. The development environments of the projects for separate components were similar since they shared the same management, were implemented on the same computer (IBM 360/65), and were of approximately the same size and duration. (For a more complete description of MIPS, see Vol. II, Section 1.3).

A decision was made at the start of the MIPS project to provide for an evaluation of a highly disciplined programming environment. Two increments of the MIPS project were selected for increased attention and measurement; the Data Analysis Processor using conventional programming techniques and the Launch Support Data Base implemented under the ASTROS plan. The arrangement allowed a quasi-experimental evaluation of the modern programming practices specified in the ASTROS plan.

1.3.1 Launch Support Data Base (LSDB). The LSDB Computer Programming Configuration Item (CPCI) is a non-real-time increment of the MIPS system which determines in advance the range characteristics and possible trajectories of missiles. LSDB represents the reprogramming of an earlier system (VIPARS) which was not referenced during the development. Since LSDB was a redevelopment, the customers were knowledgeable of its function before it was implemented. LSDB includes both data management functions and complex scientific calculations which are run in batch prior to launch operations without real-time constraints. LSDB is composed of five major Computer Programming Components (to be described hereafter as subsystems), each with numerous subroutines and procedures. LSDB was developed under the guidelines of the ASTROS plan.

1.3.2 Data Analysis Processor (DAP). The DAP reprogramming effort was a sister project to LSDB. DAP analyzed data using parameters generated by LSDB and developed reports for the MIPS system. While the ASTROS plan was not implemented on the DAP project, the management controls and development environment were made as identical as possible to those employed in the LSDB project to provide more valid comparisons between them. The DAP project was implemented in standard FORTRAN, in a non-structured coding environment, by a group of programmers not organized into a

chief programmer team.

1.4 Project Environment

The processor used in the LSDB project was an IBM 360/65 (768K, 2Mbytes LCS) which was chosen for its numerous and versatile tools. The developmental system was identical to the target system, requiring no conversion effort. The system was available for remote batch and batch use. Turnaround time for batch work was approximately 24 hours and for remote batch was approximately 2 hours. The operating system was IBM OS/360, MVT, release 21 with HASP.

After careful evaluation, S-Fortran was chosen for use on this project. Most of the LSDB code was written in S-Fortran. Small segments were coded in the BAL assembly language. S-Fortran is a high level language (Caine, Farber, and Gordon, 1974) which allows programmers to use structured concepts in their code (Dijkstra, 1972). S-Fortran did allow what Dijkstra (1972) would consider unstructured constructs. For instance, the UNDO statement allowed exits from loops. A structured precompiler converted the S-Fortran code to standard Fortran for compilation by the standard ANSI-Fortran compiler. The LSDB project had access to a subroutine library for some of the routines needed.

The project personnel underwent a series of courses designed to provide the training necessary to implement the advanced programming techniques specified in ASTROS. The trainer had studied with Yourdon, Inc. prior to teaching these courses. The curriculum included coursework in structured design and coding, program support libraries, measurement reporting, and management of structured projects.

Three types of data were collected on the LSDB project: environment data, personnel data, and project development data. Environment data provided information regarding the system such as the processor, estimated costs, and the amount of source code. Personnel data was information about the people working on the project and their evaluations of different aspects of it. In order to ensure privacy, strict anonymity was maintained and no evaluation of personnel qualifications was made. Project development data was information describing the development and progress of the project. This information included run reports, manpower loadings, and development information. Volume II - Appendix A contains the data collections forms used during the LSDB project. In addition, RADC has collected development data over a large number of systems, including military and commercial software projects (Duvall, 1978; Nelson, 1978). These data were collected in an attempt to establish

baselines and parameters typical of the software development process. These data provide a source of comparison for LSDB and DAP data.

2. Findings

Data analyses will be reported in two sections. The first section reports comparative analyses between LSDB and DAP. The order of presentation will proceed from analyses at the project level to detailed analyses of the source code. The comparisons will include:

- 1) the descriptive data generated from both projects,
- 2) the technology levels as determined from Putnam's model,
- 3) the efforts involved in generating the code as determined from Halstead's theory,
- 4) the complexity of the control structures of the systems as indexed by McCabe's metric,
- 5) the quality of the delivered source codes indicated by software quality metrics.
- 6) comparison to projects in the RADC database.

The second section of results will present analyses of the error data collected on the LSDB project. These analyses will include:

- 1) descriptive data for run-error categories,
- 2) comparison with error categories from TRW data,
- 3) trends in run-errors across time,
- 4) prediction of post-development errors.

There were some seeming discrepancies in the data. While run-error reports were obtained for the LSDB project over a 16 month period, man-hour loadings were only reported for 14 months. This discrepancy appears even greater in that numerous runs were reported during the requirements and preliminary design phases of the project, although no hours were logged to coding. Some of the initial activity on the LSDB project was performed on a component which was not peculiar to the LSDB system. Some early man-hours appear to have been charged to an account other than LSDB. Further, during the preliminary design, S-Fortran listings were used instead of a program design language. Thus, results of analyses involving man-hours should be interpreted as approximate rather than exact.

2.1 Comparative Analyses: LSDB Versus DAP

2.1.1 Descriptive data. The source code for the LSDB Computer Program Configuration Item contained over three times as many source lines and almost two and a half as many executable lines as did that of DAP (Table 1). Executable lines represented the source code minus comments, data statements, and constructs not strictly applicable to the algorithm. ENDIF, READ, and WRITE statements were not counted as executable lines. The 16,775 source lines in LSDB were distributed across five subsystems which typically ranged from 1700 to 2700 lines, with the exception of one 8013 line subsystem. The six subsystems constituting DAP were all smaller, ranging from 200 to 1400 lines of code.

Comments accounted for a larger percentage of the LSDB source code (38%) than of the DAP code (22%). While there were fewer executable lines than comment lines (31% vs. 38%, respectively) in the LSDB code, there were almost twice as many executable lines as comment lines (44% vs. 23%) in the DAP code.

The LSDB project required approximately 8081 man-hours of effort to complete, while the DAP project required approximately 6782 man-hours (Table 1). Thus, while LSDB contained 239% more total source lines of code and 140% more executable lines than DAP, the LSDB project required only 19% more man-hours of effort to complete.

Figure 3 presents a plot of the man-hours expended on LSDB and DAP during each month of the project from the requirements phase through development testing and system documentation. Different profiles were observed across the 14 months required for each of the two projects. For LSDB, the largest loadings occurred during the initial five months of the project. With the exception of month 8, only between 300 and 600 man-hours of effort were expended during each of the last nine months of the LSDB project. Man-hours expended on DAP, however, were greatest during the final months of the project. That is, during the initial 8 months of DAP only 150 to 450 man-hours were expended per month, while during the last six months (except for month 13) man-hour expenditures ranged between 500 and 1050.

The percentages of man-hours expended during each phase of development for LSDB and DAP are presented in Figure 4. On both projects, approximately 20% of the man-hours were expended in coding and integration, while 15% were expended in product documentation and training. Differing profiles of effort expenditure were observed on the other four phases for

Table 1

Comparison of Descriptive Data from LSDB and DAP

Variable	LSDB	DAP
Lines of code		
Total	16,775	4,953
Non-comment	10,413	3,860
Executable	5,205	2,169
Man-hours	8,081	6,782

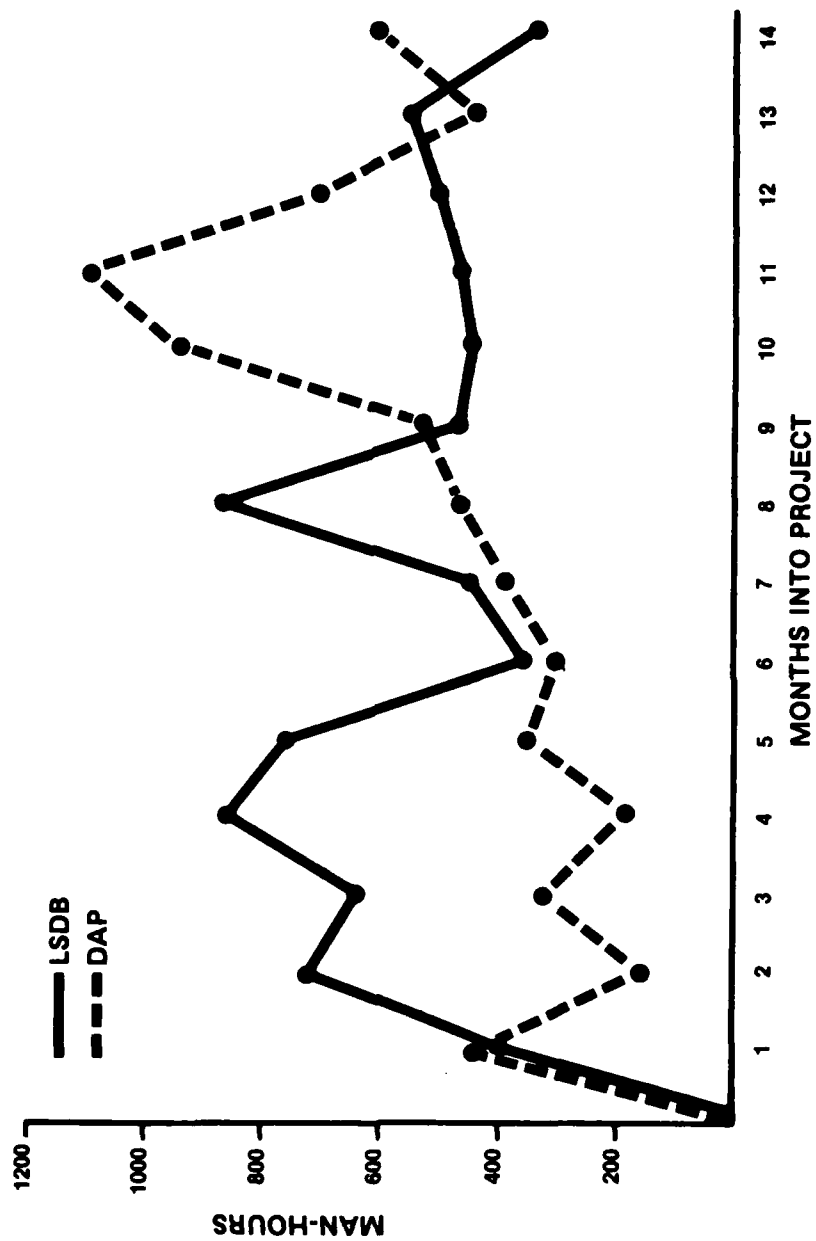


Figure 3. Chronological manpower loadings by project

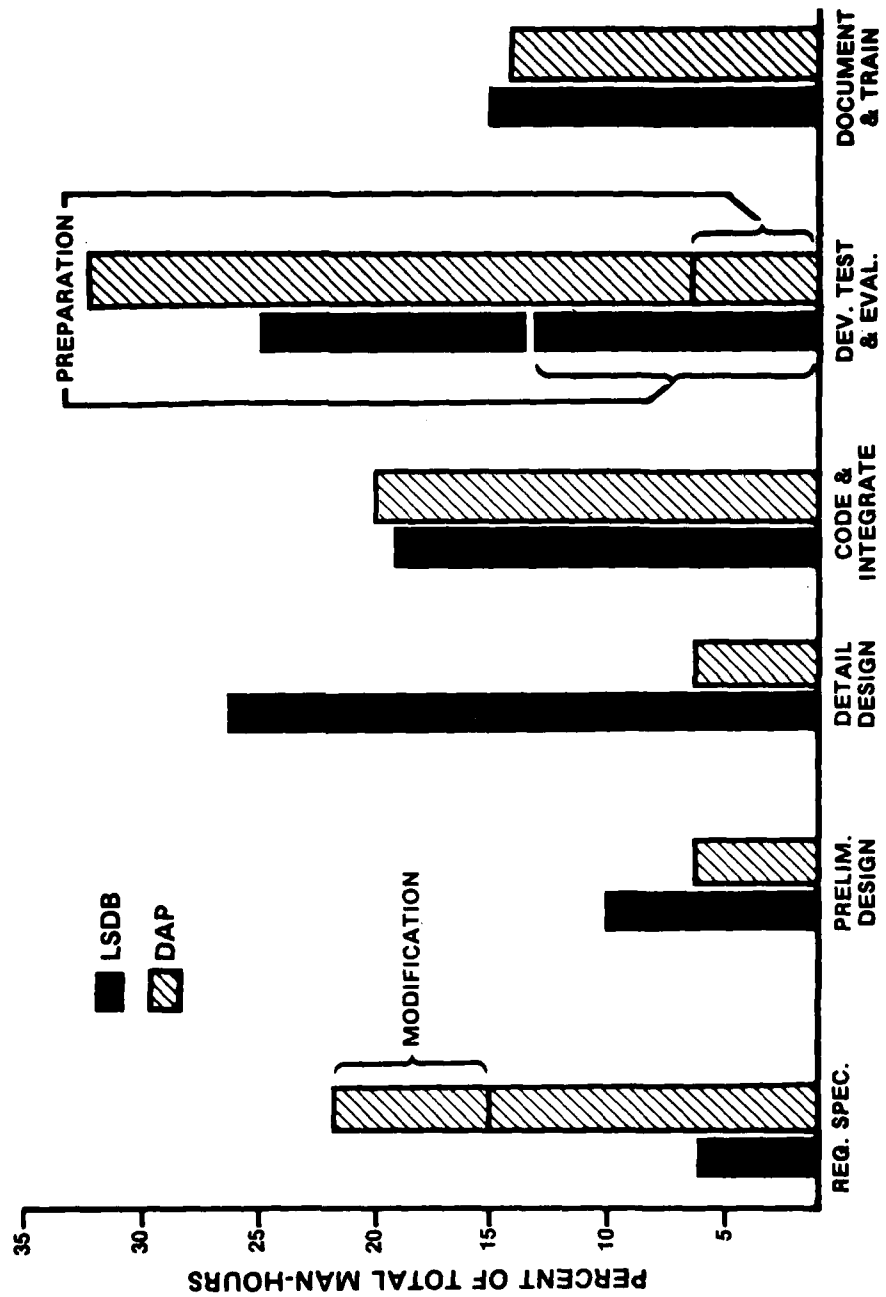


Figure 4. Percentage of total effort expended in each phase of development

each project. While only 6% of the man-hours on LSDB were invested in requirements analysis, 21% of those on DAP were consumed during this phase. However, it is likely that many of the man-hours devoted to LSDB requirements analysis may not have appeared among the manpower loadings charged to this project. Almost one third of the requirements hours for DAP was expended in what appears to have been a modification of the requirements during month 10 (Vol. II, Appendix E). When compared to the DAP project, the percentage of total man-hours on the LSDB project invested in the preliminary design was almost twice as great, and in the detailed design was over four times as great. One quarter of the man-hours on the LSDB project were expended in development testing and evaluation, while one third of those on the DAP project were so expended. However, half of the testing related time on the LSDB project was invested in preparation of the test procedures, compared to only one eighth of the DAP test-related time being devoted to preparation.

2.1.2 Level of Technology: Putnam's Model

2.1.2.1 Theory. Lawrence Putnam (1978) has refined and verified a software cost and risk estimation model which interrelates manpower, effort, development time, level of technology, and the total number of source statements in a system. His equation describes the behavior of software projects consisting of over 25,000 lines of code and must be calibrated for each development environment. When provided with initial parameters from the software project, Putnam's model can predict the manpower-time tradeoff involved in implementing the system under development. When size and time are known as in the LSDB project, the model can be applied in retrospect to obtain a technology constant. This constant reflects factors in the development environment such as software engineering techniques (e.g., modern programming practices) and hardware. These factors will determine the time and manpower needed for development. (See Vol. II; 2.1 for a more complete description of Putnam's model).

Putnam has calibrated his technology constant from analyses of data from approximately 100 systems collected by GE, IBM, TRW, and several federal agencies. In development environments typical of 10 to 15 years ago, where programming was performed in batch mode and written in assembly language, values could be as low as 1,000. Development environments of systems built with a higher order language such as FORTRAN, in batch processing, on one large mainframe saturated with work, and slow turnaround could yield technology constants of around 5,000. Higher values occurred in an environment where modern programming practices were implemented with on-line,

interactive programming.

2.1.2.2 Results. Putnam's technology constant for the LSDB project was anticipated to fall in the midrange of values (approximately 5,000) because advanced tools were employed, but the computer was used in batch mode, turn around time was slow, programming was non-interactive, and some of the code was written in assembler. The computed value of 6,685 is slightly higher than anticipated. The level of technology computed for DAP was 2,891. This figure is lower than the technology constant computed for the LSDB project. Since both projects used the same machine and access times were similar, differences in the technology constant cannot be attributed to a factor Putnam considers to be one of the primary influences on his metric; access time. Although Putnam suggests that his equation will yield an overestimate of the technology constant for projects of under 25,000 lines of code, these computations nevertheless validate the ASTROS plan as providing a more advanced programming technology than the conventional techniques practiced on the DAP project.

2.1.3 Programming Scope and Time: Halstead's Model

2.1.3.1 Theory. Maurice Halstead (1977) has developed a theory which provides objective estimates of the effort and time required to generate a program, the effort required to understand a program, and the number of bugs in a particular program. In 1972, Halstead first published his theory of software physics (renamed software science) stating that algorithms have measurable characteristics analogous to physical laws. According to his theory, the amount of effort required to generate a program can be calculated from simple counts of the actual code. The calculations are based on four quantities from which Halstead derives the number of mental comparisons required to generate a program; namely, the number of distinct operators and operands and the total frequency of operators and operands. Preliminary tests of the theory reported very high correlations (some greater than .90) between Halstead's metric and such dependent measures as the number of bugs in a program, programming time, and program quality.

Halstead presents a measure of program size which is different from the number of statements in the code. His measure of program volume is also independent of the character set of the language in which the algorithm was implemented. Halstead's theory also generates a measure of program level which indicates the power of a language. As the level approaches 1, the statement of the problem or its solution becomes more succinct. As the program level

approach 0, the statement of a problem or its solution becomes increasingly bulky, requiring many operators and operands. A higher level language is assumed to have more operators available, but these operators are more powerful and fewer operators need to be used to implement a particular algorithm. Halstead theorized that the effort required to generate a program would be a ratio of the program's volume to its level. He proposed this measure as representing the number of mental discriminations a programmer would need to make in developing the program. The time required to generate the program could be estimated by dividing the effort by the Stroud (1966) number of 18 mental discriminations per second. (See Vol. II; 2.2.1 - 2.2.3).

In using Halstead's equations to compute the effort or time required to develop a system, it is important to limit the computations to the scope of the program that a programmer may be dealing with at one time. By scope, we mean the portion of a program that a programmer is attempting to represent to himself cognitively, regardless of whether he is developing code or attempting to understand existing code. There are several strategies that a programmer could follow while working on a module, and they result in different values for the overall effort. (See Vol. II; 2.2.4 - 2.2.4.2.2).

In the minimum scope case, the programmer would only keep the context of the subroutine he was currently working on in mind. He would not keep track of other variables from other routines. In this case, each subroutine could be considered a separate program, and the effort for the subsystem would be the summation of the effort for the separate subroutines.

In maximum scope cases, the programmer is assumed to treat the entire subsystem as one program. He mentally concatenates the subroutines into a subsystem and treats the subsystem as one complete algorithm (a gestalt) where he must keep track of all its aspects at any given time. Data presented in Vol. II; 3.1.3 indicate that the minimum scope case where a programmer concentrates on one module at a time is more consistent with the empirical evidence. Therefore, only results for this case will be presented.

2.1.3.2 Results. The total volume of the LSDB code is greater than that of DAP (Table 2). Further, the effort for LSDB was less and the level greater than that for DAP. According to the Halstead model, the effort required to code LSDB was less than that required to program DAP. This may have occurred in part because the programming level evident in the source code was greater for LSDB. That is,

Table 2
Comparison Between LSDB and DAP on Halstead's Metrics

Metric	LSDB	DAP
Volume	249793	131587
Level	.0082	.0046
Effort	36.93M	71.97M
Productivity predicted from other project	3639	565
Actual coding time	1524	1349
Predicted coding time	570	1111

the LSDB code appears to have been written more succinctly than the DAP code. This comparison becomes even more evident if we use the productivity defined as non-comment lines produced per man-hour from each project to predict the programming time for the other project. If the rate of non-comment lines produced per man-hour for DAP had been true for the LSDB project, it would have taken almost two and one half times as many man-hours to produce the non-comment portion of the LSDB code.

Finally, the predictions from the Halstead model underestimate the actual times required to code the two CPCI's, and in the case of LSDB this underestimation is considerable. This underestimation agrees with the reports of LSDB project programmers that they had considerable slow time during the coding phase due to poor turnaround time.

2.1.4 Complexity of Control Flow: McCabe's Model

2.1.4.1 Theory. Thomas McCabe (1976) defined complexity in relation to the decision structure of a program. He attempted to assess complexity as it affects the testability and reliability of a module. McCabe's metric, $v(G)$, counts the number of basic control path segments through a computer program. These are the segments which when combined will generate every possible path through the program. McCabe's $v(G)$ can be computed as the number of predicate nodes plus 1 or as the number of regions in a planar flowchart of the control logic.

McCabe also presents a method for determining how well the control flow of a program conforms to structured programming practices. He decomposes the control flow by identifying structured constructs (sequence, selection, and repetition) that have one entrance and one exit and replaces them with a node. If a program is perfectly structured, then the flow graph can be reduced to a simple sequence with a complexity of 1 by iteratively replacing structured constructs by nodes. If a segment of code is not structured, then it cannot be decomposed and will contribute to the complexity of the program. McCabe calls the ultimate result of this decomposition the essential complexity of the program (See Vol. II; 2.3).

2.1.4.2 Results. In order to evaluate the complexity of the control flows using McCabe's procedures, two subroutines were selected for analysis from the source code of each project. McCabe's $v(G)$ was computed on each subroutine and the values ranged from 76 to 165 (Table 3). However, differences in the number of executable statements in each

Table 3

McCabe's v(G) and Software Quality Metrics for
Selected Subroutines from LSDB and DAP

Metric	<u>LSDB</u>		<u>DAP</u>	
	1	2	1	2
<u>McCabe's v(G)</u>				
Actual	76	136	147	165
Adjusted	101	119	114	132
<u>Essential v(G)</u>				
Actual	31	18	50	56
Adjusted	41	18	39	45
<u>Software quality metrics</u>				
Structured language	1.00	1.00	.00	.00
Coding simplicity	.67	.69	.63	.57
Modular implementation	.55	.41	.00	.00
Quantity of comments	.49	.28	.46	.17
Effectiveness of comments	.45	.45	.43	.40
Descriptiveness of implementation language	1.00	.98	.83	.83

subroutine over which these values were computed made their comparison difficult. Values of $v(G)$ for each subroutine were adjusted proportionately to a value for a standard subroutine of 300 lines of code. Following this transformation, the control flows of the subroutines from each project were found to be of approximately equal complexity.

The essential complexity of the four modules ranged from 31 to 56 (Table 3). The adjusted scores indicated that one of the subroutines from LSDB was similar to those from DAP in its degree of unstructuredness, while the other was substantially more structured. Although this unstructuredness in LSDB seems surprising, S-Fortran allows an UNDO statement, which results in an unstructured construct by allowing jumps out of loops (Figure 5). With the exception of this construct, the LSDB code in the two modules analyzed was consistent with the principles for structured code described by Dijkstra. The unstructured practices in the DAP code were much more varied.

2.1.5 Software Quality Metrics

One of the most comprehensive studies of software quality was performed by McCall, Richards, and Walters (1977) under an RADC contract. They defined various factors of software quality as a means for quantitative specification and measurement of the quality of a software project. They developed a set of metrics for evaluating each of their factors on a unit of code. With these aids, a software manager can evaluate the progress and quality of work and initiate necessary corrective action.

Several software quality metrics concerning modularity, simplicity, and descriptiveness reported by McCall, Richards, and Walters, (1977) were computed on the subroutines used in the McCabe analysis. (The detailed scoring of the software quality metrics relevant to this study can be found in Vol. II Appendix D). Table 3 summarizes the results of this analysis. On two of the six measures studied the LSDB code was found to be clearly superior to the DAP code. That is, LSDB was written in a language which incorporated structured constructs, and the system design was generally implemented in a modular fashion. The LSDB code in the subroutines analyzed deviated from the principles of top-down modular design to the extent that calling procedures did not define controlling parameters, control the input data, or receive output data. The LSDB subroutines received slightly higher scores for coding simplicity than those of DAP. These scores reflected better implementation in LSDB of top to bottom

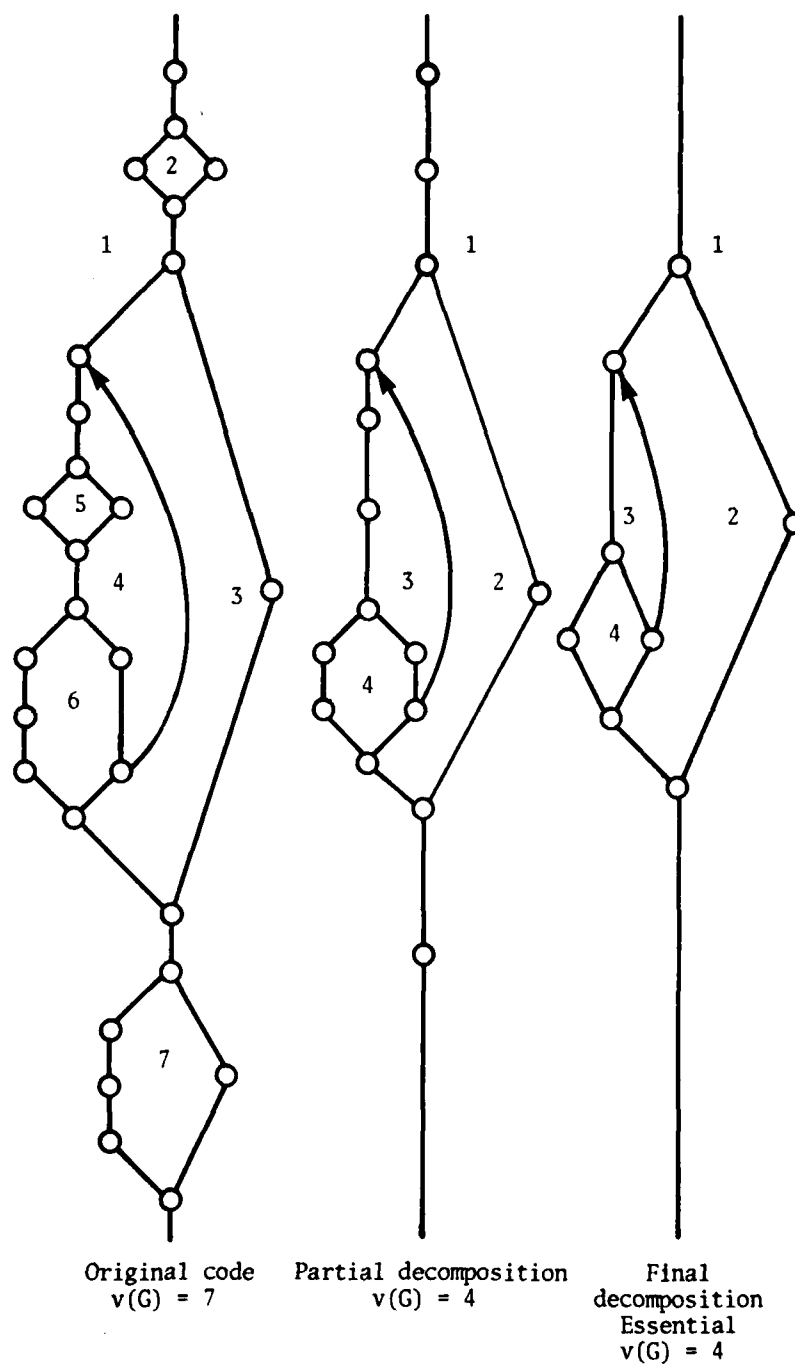


Figure 5. Decomposition to essential complexity

modular flow, use of statement labels, and avoidance of GO TO's. Minor differences were observed between projects on the effectiveness of comments. Thus, the greater percentage of comments in the LSDB code may not have contributed proportionately more to the quality of documentation beyond the quality observed in DAP. The descriptiveness of the LSDB code (e.g., use of mnemonic variable names, indentation, single statement per line, etc.) was slightly greater than that of the DAP code.

2.1.6 Comparison to RADC Database

Richard Nelson (1978) has performed linear regressions of delivered source lines of code on other variables in the RADC software development database. These regressions allow an investigation of performance trends as a function of project size. When outcomes from the LSDB and DAP projects were plotted into these regressions, it becomes possible to compare the performance of LSDB and DAP with other software development efforts while controlling for the size of the project (in terms of delivered source lines). Figure 6 presents the scatterplot for the regression of delivered source lines of code on productivity (lines of code per man-month). The datapoints for the LSDB and DAP projects fall within one standard error of estimate of the regression line. However, LSDB falls above the regression line and DAP falls below it, suggesting that LSDB's productivity was slightly higher than the average productivity for projects of similar size and DAP's was slightly lower. Scatterplots presented in Vol. II, Appendix B for regressions of delivered source lines on total man-months, project duration, total errors, error rate, and number of project members indicated similar results. That is, the performance of LSDB was usually better than that of DAP when compared to projects of similar size. However, the data points for both usually fell within one standard error of estimate of the predicted value.

2.2 Error Analyses

All of the data reported in this section are from the LSDB project with the exception of the post-development errors for which data were available from DAP. No record of development runs were available from DAP.

2.2.1 Error Categories

2.2.1.1 Descriptive data. Of the 2,719 computer runs involved in the development of LSDB, 508 (19%) involved an error of some type (Vol. II; Appendix F). The frequencies of these errors are reported in Table 4 by a categorization scheme similar to that developed by Thayer et al. (1976) for

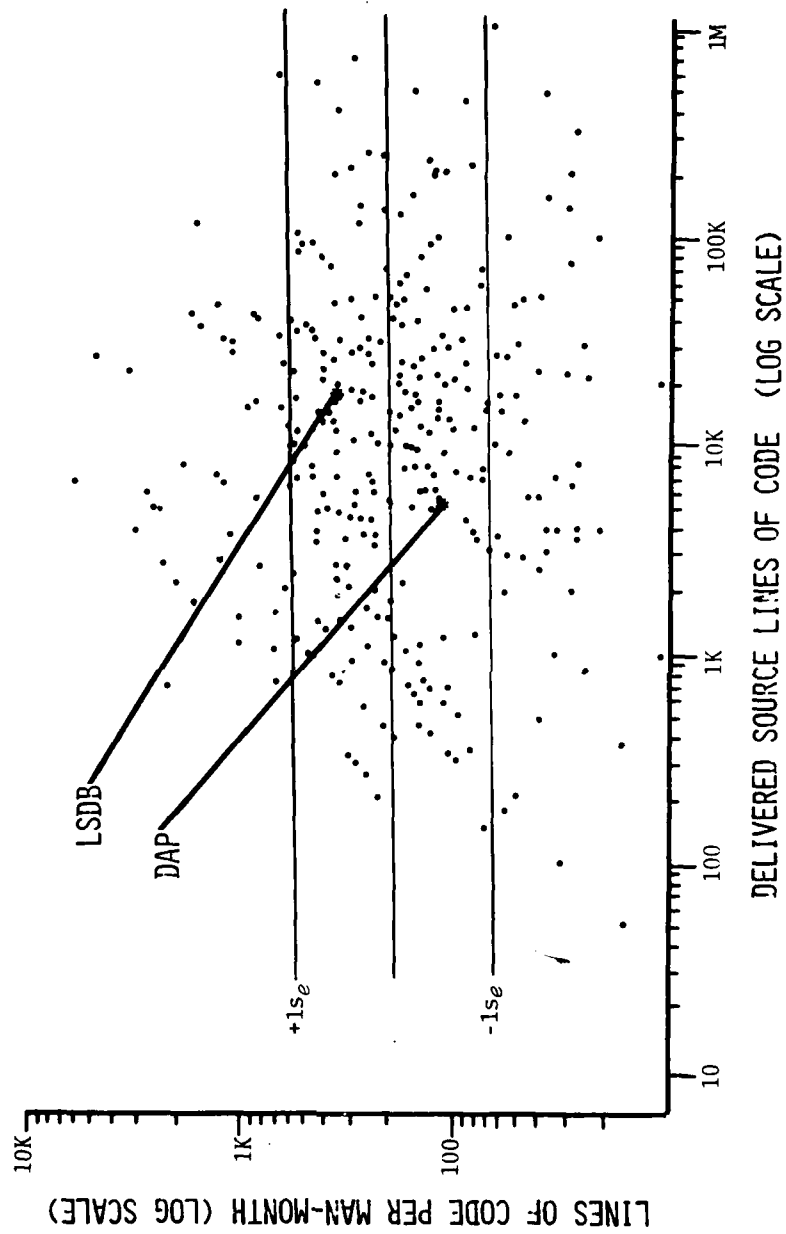


Figure 6. Scatterplot for the regression of delivered source lines of code on productivity

Table 4
Frequencies of Error Categories

Error category	f	%
<u>Algorithmic errors</u>		
Computational	5	1
Logic	98	19
Data input	17	3
Data handling	12	3
Data output	3	1
Interface	0	0
Array processing	1	0
Data base	4	1
Total	140	28
<u>Non-algorithmic errors</u>		
Operation	115	23
Program execution	41	8
Documentation	0	0
Keypunch	51	10
Job control language	73	14
Total	280	55
<u>Unclassified</u>	88	17
<u>Grand total</u>	508	

TRW Project 5. The 14 categories of errors are divided into two groups and a lone category of unclassified errors. The first group involved errors which were detected by a malfunction of the algorithm implemented in the code. These categories included computational errors, logic errors, data errors, etc., and accounted for 28% of the total errors recorded. Most numerous were logic errors which constituted 19% of all errors.

Over half of the errors recorded (55%) were non-algorithmic and involved the hardware functioning (23%), the job control language (14%), keypunch inaccuracies (10%), or program execution errors (i.e., compile error or execution limits exceeded, 8%). Seventeen percent of the errors recorded could not be classified into an existing category in either group.

2.2.1.2 Comparison with TRW data. As a partial test of the generalizability of these error data, the profile across selected error categories was compared to similar data from three development projects conducted at TRW (Thayer et al., 1976). Data are reported only for those errors for which similar classifications could be established. This analysis was performed and first reported by Hecht, Sturm, and Trattner (1977). The percentages reported in Figure 7 were developed by dividing the number of errors in each category by the total number of errors across the five categories. LSDB and the fourth study from the TRW data were found to be quite similar, especially with regard to the low percentage of computational errors and the high percentage of logic errors. The LSDB error profile was similar to the other two TRW studies in the percent of data input and handling errors and interface/program execution errors. Overall, it would appear that the distribution of error types on the LSDB project is similar to distributions observed on other projects.

2.2.2 Error Trends over Time

A chronological history of the number of action reports, error-runs (total and algorithmic), and post-development errors is presented by month in Vol. II, Figure 12. The 63 action reports describing discrepancies between the design and the functional requirements were recorded only during the first seven months of development. Error-runs were reported over the entire 16 months of development. The distribution of total errors over months was bi-modal, with the first mode occurring during the second and third months of the project, and the second mode occurring between the ninth and eleventh months of the project. Post-development

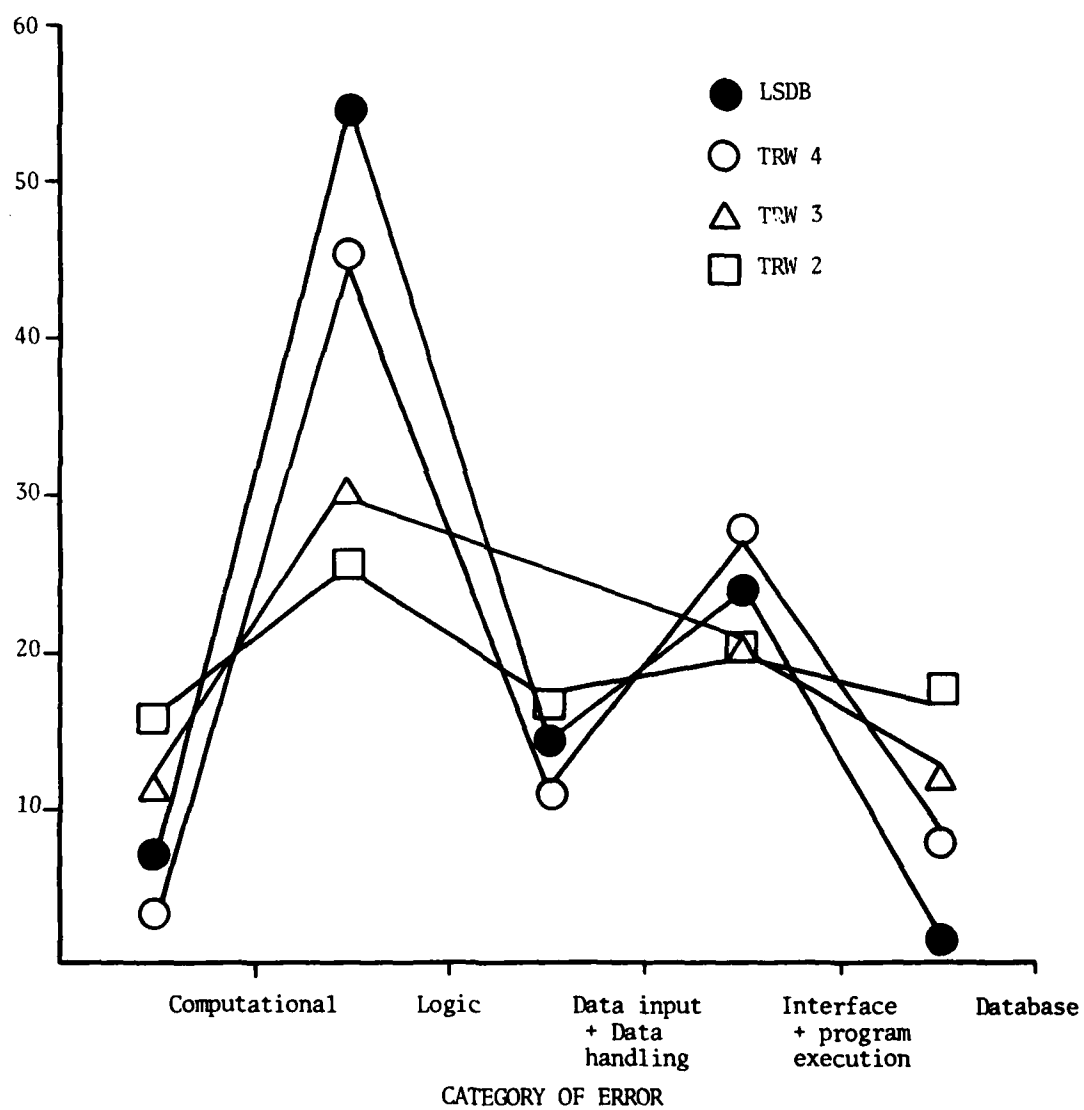


Figure 7. Comparison of error distributions between LSDB and three TRW studies

errors were first recorded during month 16 and continued until month 24. However, no activity was recorded from months 17 to 19.

The development error data indicated that work was performed on all subsystems nearly every month through the life of the project. These data suggest that the project team proceeded with the parallel development of subsystems. The alternative approach of depth first coding and implementation where one subsystem is completed before proceeding to the next, did not appear to have been employed.

Since the number of errors per month varied with the number of runs, a more representative measure of error generation was developed by dividing the number of errors by the number of runs. These rates for both total and algorithmic errors are plotted by month in Figure 8. Linear regressions for these error rates indicated decreasing trends over time. For total errors the correlation was $-.52$, while for algorithmic errors it was $-.63$. Rates for total errors decreased sharply over the final nine months from 30% in month 8 to 8% in month 16. The correlation associated with this sharp decline was $-.90$ (predicted error rate = $-2.5(\text{month \#}) + 47.78$).

2.2.3 Post-Development Errors

There were 28 post-development errors reported for the LSDB code, and their frequency declined over time. These errors included 12 subsystem development test errors and 16 system integration test errors. Forty-three system integration test errors were reported for DAP. Compared to the size of the source code, proportionately fewer post-development errors were reported for LSDB. This comparison is even more striking because reports of subsystem development test errors were not available for DAP, thus the total number of post-development test errors for DAP should be even larger.

There are several methods of predicting the number of post-development errors from the kinds of data available here, the results of which are presented in Figure 9. Halstead's (1977) equations for the total number of delivered bugs (see Vol II; 2.2.5) led to a prediction of 27.2 errors for LSDB and 4 errors for DAP. Thus, the prediction was amazingly accurate for LSDB and substantially underestimated for DAP. Since much of Halstead's original work was performed on published algorithms, the accuracy of his predictions may improve with the quality of the code and the extent to which the code is an accurate reflection of the requirements and specifications. Such an explanation would

— TOTAL ERRORS
 - - - ALGORITHMIC ERRORS

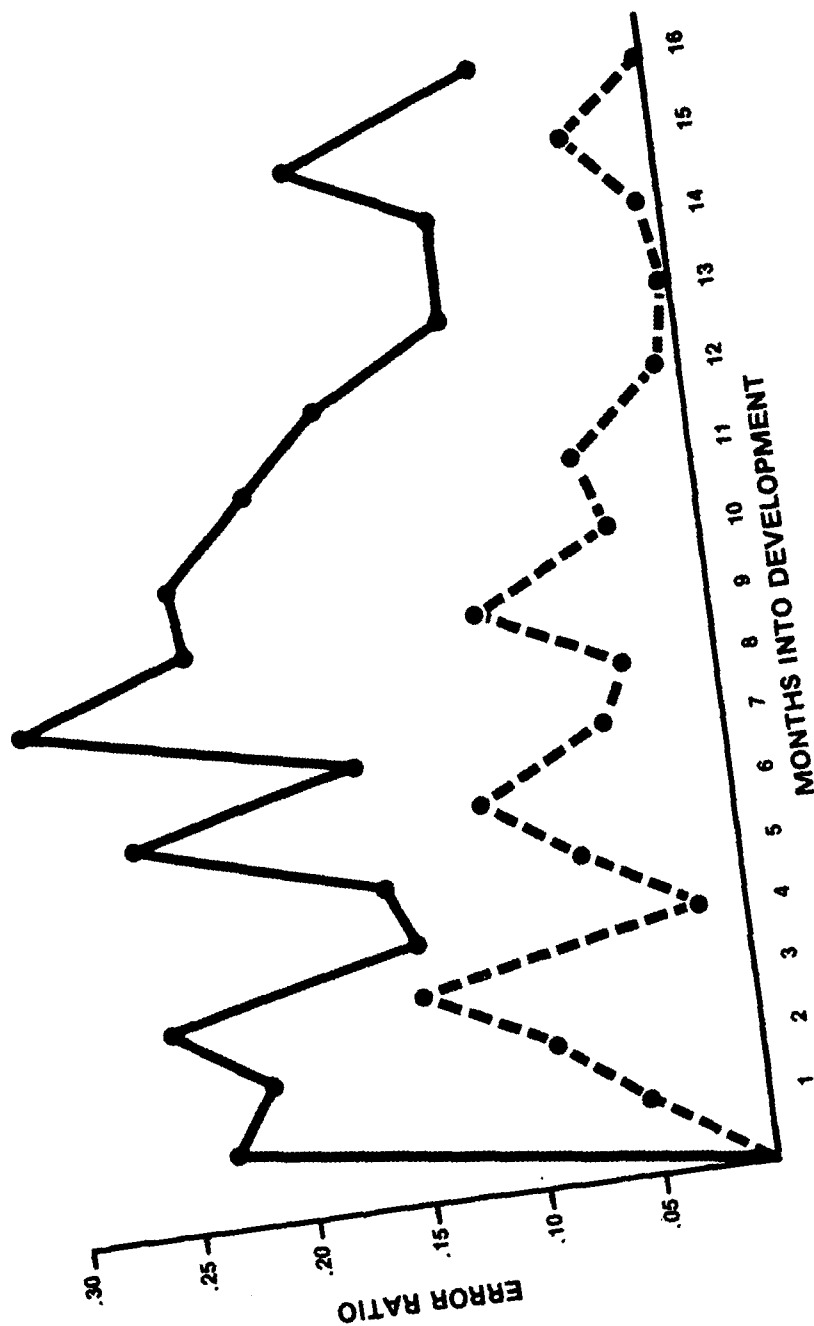


Figure 8. Error ratio by month

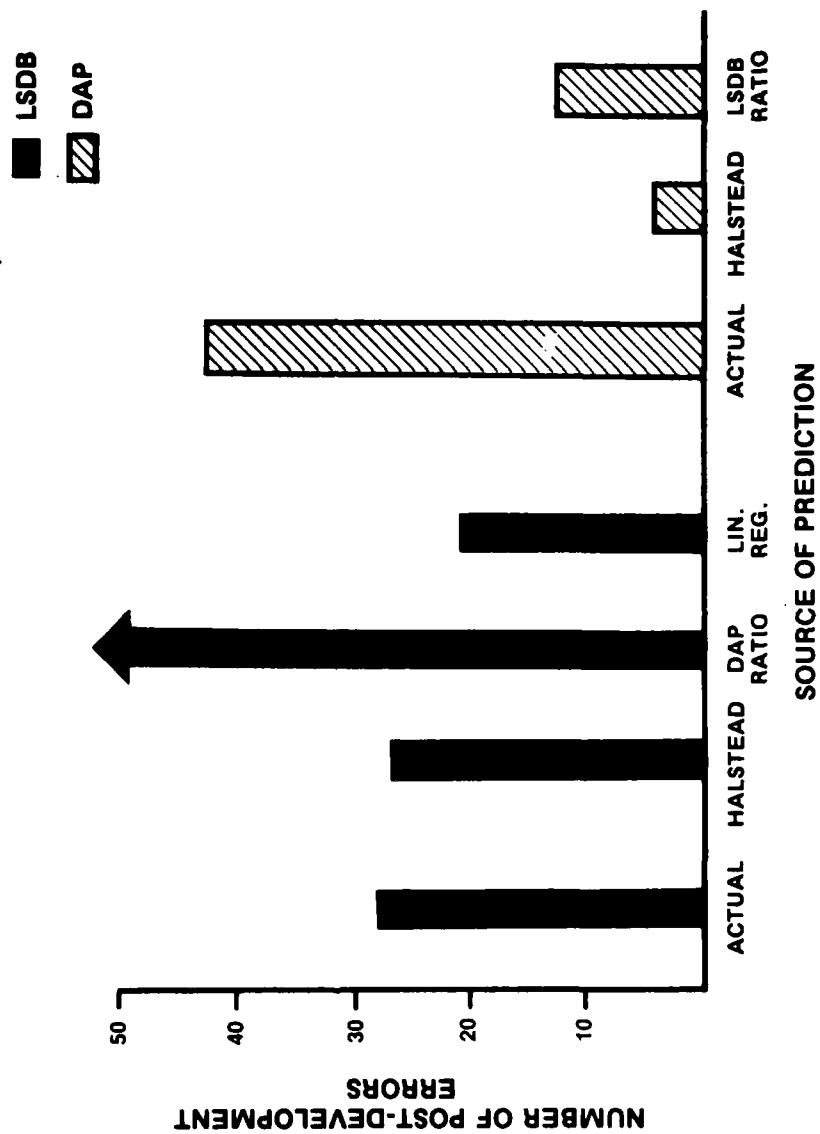


Figure 9. Prediction of post-development errors

be consistent with observations in these data, and with the fact that some requirements redefinition appears to have occurred during the DAP project.

The ratio of post-development errors to number of executable lines of code from each project was used to predict the number of post-development errors in the other project. The ratio for LSDB was 0.0027 and for DAP was 0.0111. When the ratio for the LSDB project was multiplied by the number of executable lines in DAP, it led to a prediction of only 10 post-development errors for the DAP project (Figure 9). When the ratio for DAP was multiplied by the number of executable lines in LSDB, it led to a prediction of 116 post-development errors for LSDB. Thus, it is obvious that the proportion of post-development errors per line of executable code was four times greater for DAP than for LSDB.

An attempt was made to predict post-development errors from error-runs occurring in the development work performed prior to testing. As a simple heuristic in place of curve-fitting, a linear prediction equation was developed from the error rates occurring between months 8 and 15, since the standard error of estimate for error rates over this period was much smaller. (See Vol II; 3.2.4 for a more complete description of the method). The equation developed from the error rates recorded during months 8 through 15 was $-2.52X + 43.02$, where X represents the month number. It was predicted that 22.5 post-development errors would be detected. This estimate is reasonable close to the 28 post-development errors actually reported.

3. INTERVIEWS

The following general observations emerged from interviews with individuals associated with the LSDB project. On the basis of these interviews it appeared that the success enjoyed by this programming team was partly achieved by a fortuitous selection of team members. The particular personal styles of members were well-suited for their roles on the team and for the practices employed. The observations gleaned from these interviews are discussed below.

3.1 Chief Programmer Teams

The team members interviewed felt that the most significant practice was the chief programmer team. In particular, they felt that the cohesiveness and the visibility of each member's work contributed to higher performance. It took several months of interaction before the team really became cohesive. The team felt that a strong source of morale resulted from keeping the nucleus of the team intact throughout the development cycle (contrary to ordinary practice at SAMTEC), with the possibility of continuing into future assignments. Since all team members shared in most tasks, no job (with the possible exception of librarian) was allowed to become too tedious or mundane.

Project participants felt the team needed its own physical space. The LSDB team had their own room with a lock on the door. The team worked around a conference table resulting in continuous interaction among team members, and ensured that the work of each member was consistent with the work of others. During afternoon breaks, the team would make popcorn, but some constraints were placed on their independence when an administrator found them making ice cream on a side porch.

The chief programmer was a software craftsman primarily interested in doing technical work without being hampered by administrative, personnel, or bureaucratic problems. He was an experienced software designer and coder who, because he understood the technical characteristics of the target system better than other project members, had the final say on system design and coding. Both the chief programmer and the tester were considered "range rats" (a local term for people who had considerable experience working on the Western Test Range at Vandenberg AFB). Over the years they had performed many jobs at Vandenberg and could anticipate many of the practical and technical problems they encountered during the software development. The chief programmer was a dynamic individual who was able to establish close working relationships with his customers.

The backup programmer had to work very closely with the chief programmer and their skills were supplementary. The backup programmer was quieter, a hard worker, and capable of generating large amounts of code. The backup did 80% of the coding for the system.

The librarian was responsible for taking the code and entering it into the system. The librarian was the only person who could actually change or add code to the system, although other members might have access to add data. The librarian was able to tolerate time constraints and maintain careful control throughout the project. The librarian was more than a keypuncher or "go-for", and was expected to develop some basic technical abilities such as setting up job control. The job level of the librarian was one step higher than an administrative specialist and several steps higher than clericals.

The procedure for submitting programs in this project required either the chief programmer or the backup to submit a written version of the program or change to the librarian, who would enter it into the system, perform an initial check, eliminate any obvious errors, and then return a compiled listing of the program to the backup programmer. The backup or chief programmer would review this listing before performing the first execution. Careful records were kept of every submission, any resulting errors, and the code that had to be changed.

The tester worked along side the team almost as an alter ego. He set up the necessary data for both intermediate and final acceptance tests. It was important that the tester was not considered a member of the team. Nevertheless, the tester was not someone brought in at the last minute. From project initiation, he attended all the walk-throughs and became intimately familiar with the LSDB code.

Team members felt it was important that the team be able to select its members. Similarly, they felt the team should be able to oust members who were not compatible or who were not contributing to team efforts. The following list summarizes the team's recommendations for selecting members. The selection process should eliminate only incompetent programmers. A chief programming team represents a particular mix of skills and duties which may not be acquired if only brilliant programmers are selected (someone has to slog through the trenches). Some particular characteristics which seemed important to this team were:

Chief Programmer

- dynamic
- excellent technical proficiency
- delegates red tape to someone higher up
- can establish close working relationships with customers
- has an almost paternal, protective (but democratic) attitude toward the team

Backup Programmer

- less dynamic
- areas of technical competence supplement those of Chief Programmer.
- limitless capacity for generating code
- should be capable of filling in for Chief Programmer when necessary

Support Programmer

- willing to participate in team interactions and work within team consensus
- cannot be a prima donna, a solitary worker, or unable to take criticism
- should be given a trial period before becoming a formal team member

Librarian

- high tolerance for frustration and pressure
- willing to perform unending work
- needs some understanding of programming code
- needs typing skills

Tester

- needs experience in content area of the program
- must be assertive
- should not become close knit with team

The team felt that new members would need some orientation to the working environment they were entering. Such training might be most effectively handled by experienced chief programmers rather than professional trainers. The type of people chosen for a team should be those who are adaptive to the types of new habits that will be the focus of training (this may be easier for less experienced programmers).

3.2 Design and Coding Practices

LSDB was designed in a strictly top-down fashion and this practice was considered an important contribution to the success of the system. The chief programmer commented that in his 20 years of experience he had usually done top-down design, but that he had not employed top-down development. He considered them both to be important, especially when used in tandem. However, he reported problems in trying to implement the top-down development strategy within standard Air Force practices, especially the requirements for preliminary and critical design reviews. Considerable time was lost while waiting for the Configuration Control Board's approval of specifications.

In an attempt to implement modular design, most procedures in the system were constrained to a maximum of 100 lines of code or two pages of output including comments. There was an attempt to make each of the subroutines as functionally independent as possible and to restrict unnecessary data transmission between modules (G. Myers, 1978). The team favored the use of highly modular systems and believed this practice contributed significantly to the ease with which the resulting system could be maintained or modified. Although they thought structured coding was of benefit, they considered its relative importance to be small compared to that of modularity or the use of the chief programmer teams.

Variations in the construction of HIPO charts from guidelines described by IBM were identified by an independent validation and verification contractor and corrected. HIPO charts were very unpopular with both the team and the independent contractor. HIPO's were not maintained up-to-date throughout the project. The team felt that the HIPO charts were not particularly useful beyond the first or second level within the system hierarchy. HIPO's might have been more readily accepted had an interactive system been available for generating them. The team recommended that a program design language (PDL) would have been much more useful than HIPO charts.

Walk-throughs were held every week and were attended by the software development team, the tester, the project administrator to whom the team reported, the customer, and the end users of the system. The team felt it was important that these walk-throughs were held weekly and that those involved in the procurement and use of the system were in attendance. In fact, since the team held code reviews internally on an ad hoc basis, they felt that walk-throughs

were held primarily for the benefit of other interested parties.

Walk-throughs tended to last one to one and one-half hours. In most cases, the chief programmer handed out either the code or design description a week prior to its consideration so that all attendees could review the material and be prepared with detailed comments. They discovered early that walking through the code in detail was not practical. Rather, they gave high level descriptions of the routine's processing and went into the code only as required.

The team felt that the amount and type of documentation required was burdensome, especially the documentation of specifications and design. Both the developers and maintainers felt that the code was sufficiently well designed and documented internally that no other documentation was required. In no case had they gone back to the HIPO charts or even the specifications to obtain information in order to make a change to the system. Sophisticated documentation may not have seemed as important since most necessary modifications were so minor that a single statement could be isolated rather quickly.

As of March 1978 very little maintenance had been required. Since much of the maintenance has been minor, it has been suggested that the team librarian or someone with equivalent experience could make most of the one card modifications that have been required. The LSDB development team and tester received a letter of commendation from the end users. They were complimented both for the high quality of the software and for its production on time and within budget.

One anecdote is instructive on the effectiveness of the ASTROS guidelines. At the beginning of the project, the LSDB team was required to suspend work while waiting for the preliminary design review. In order to keep them from beginning to code, the SAMTEC engineer gave the team a problem from another project. The program computed and plotted some range safety parameters. A tiger team had spent a month trying in vain to modify the existing program for an upcoming launch. The LSDB team redesigned the program and produced a working parameterized version in one week.

4. CONCLUSIONS

4.1 Current Results

The performance of the LSDB development project using the modern programming practices specified in the ASTROS plan was comparable to that of similar sized software development projects on numerous criteria. The amount of code produced per man-month was typical of conventional development efforts (however, this is a controversial measure of productivity; Jones, 1978). Nevertheless, the performance of the LSDB project was superior to that of a similar project conducted with conventional techniques in the same environment. Thus, the benefits of the modern programming practices employed on the LSDB project were limited by the constraints of environmental factors such as computer access and turnaround time.

While the results of this study demonstrated reduced programming effort and improved software quality for a project guided by modern programming practices, no causal interpretation can be reliably made. That is, with only two projects and no experimental controls, causal factors can only be suggested, not isolated. The ability to generalize these results to other projects is uncertain. For instance, it cannot be proven that modern programming practices had a greater influence on the results than differences among the individuals who comprised the LSDB and DAP project teams. Having acknowledged this restriction on causal interpretation, however, it is possible to weave together evidence suggesting that important benefits can be derived from the use of modern programming practices.

Several analyses demonstrated that improved efficiency was achieved through the use of the modern programming practices specified in the ASTRO plan. The value of Putnam's technology constant computed for LSDB was higher than for DAP. Further, the relative values of the LSDB and DAP projects on the parameters described in the RADC database consistently showed LSDB to have a higher performance than DAP when compared to projects of similar size, although the performance of both was close to the industry average. Since the DAP and LSDB projects shared similar processing environments, differences between the systems are probably not attributable to environmental factors.

The LSDB project demonstrated more efficient use of development man-hours than the DAP project. The Halstead parameters indicated that the LSDB project generated more code with less overall effort than DAP. LSDB exhibited a higher program level, indicating a more succinct representation of the underlying algorithm than was true for

DAP. The results of the effort analyses emphasize the great power in modular approaches to programming. If a programmer is required to keep the total context of a subsystem in mind, the time and effort required by the project increases. By breaking up the project into independent functional subroutines, the load upon the programmer is reduced.

Another area of evidence favoring modern programming practices was the superior quality of the LSDB code compared to that of DAP. Scores on the software quality metrics produced by McCall, Richards, and Walters verified that the practices employed on the LSDB project resulted in a more modularized design and structured code than DAP. Further, greater simplicity was evident in the LSDB code. Although the McCabe analysis indicated that the complexity and structuredness of the control flows were generally similar, the breaches of structured practice in the LSDB code were uniform. That is, the UNDO construct in S-FORTRAN allows branches out of conditions and loops. Although used consistently in the LSDB code, this construct is not among the structured programming practices recommended by Dijkstra (1972). This construct may not have made the control flow more difficult to understand (Sheppard, Curtis, Milliman, & Love, 1979). Departures from structured principles were far more varied in the DAP code, resulting in a convoluted control flow that is much more difficult to comprehend and trace.

Perhaps the most impressive comparison between LSDB and DAP concerns the number of post-development errors. When compared to DAP, the LSDB code contained three times as many lines and two and a half times as many executable lines, but only two-thirds as many post-development errors were reported for LSDB. The reliability achieved by LSDB was well predicted by both the Halstead equation for delivered bugs and a regression equation based on monthly error rates during development. The inability of any of the methods to predict post-development errors on DAP suggests that the prediction obtained for LSDB may have occurred by chance. Nevertheless, the existence of a tester associated with the LSDB development and the orientation of the project towards its final evaluation may have contributed to a strong correspondence between the requirements and the delivered code of LSDB. When this correspondence exists, the number of errors or error rate may prove much more predictable.

It is clear from analyses reported here that a software development project employing modern programming practices performed better and produced a higher quality product than a conventional project conducted in the same environment. However, these data do not allow the luxury of causal

interpretation. Even if such interpretation were possible, the data still do not allow analyses of the relative values of each separate practice. Further evaluative research will be required before confident testimonials can be given to the benefits of modern programming techniques. Nevertheless, the results of this study suggest that future evaluations will yield positive results if constraints in the development environment are properly controlled.

4.2 Future Approaches to Evaluative Research

This study demonstrated that exercising some control over the research environment can be extremely valuable. Without the comparable environment of LSDB and DAP this study would have found little evidence to indicate that modern programming practices have benefit. The experimental manipulation of selected practices (e.g., different ways of organizing programming teams) would improve future research efforts. Unless the separate effects of different practices can be identified, no recommendations can be made concerning them. Rather, the only conclusion that can be reached concerns the use of modern programming practices as a whole.

Data collection on programming projects often interferes with the programming task or meets opposition from team members. This problem can be counteracted by developing measurement tools embedded in the system which are invisible to programmers. Such tools would produce more reliable data since they cannot be forgotten, ignored, or incorrectly completed as manually completed forms frequently are. On the LSDB project, information such as number of runs and errors and time per run was acquired directly from the operating system. Manpower loadings can be taken from the hours recorded on time cards to be billed to the project, and can be broken down into job classifications. The program source code is also an excellent source of data concerning the quality and complexity of the code, and the number and types of statements. A program support library can keep track of the relative status of programs with minimal impact on programmers, the number of modules, and the time spent on each one.

Data collection can serve the purposes of both research and management. A software development manager needs visibility of project progress in order to control events and determine corrective action. A program support library can report module size, number of runs, and other summary information at regular intervals, and check project status to alert the manager to milestones. When these collection mechanisms are imposed unobtrusively on programming projects,

their use is more likely to gain support, and better data will be available both to management and researchers.

5. REFERENCES

- Baker, F.T. Chief programmer team management of production programming. IBM Systems Journal 1972, 11, 56-73.
- Baker, F.T., & Mills, H.D. Chief programmer teams. Datamation, 1973, 19 (12), 58-61.
- Barry, B.S., & Naughton, J.J. Structured Programming Series (Vol. 10) Chief Programmer Team Operations Description (RADC-TR-7400-300-Vol.X). Griffiss AFB, NY: Rome Air Development Center, 1975 (NTIS No. AD-A008 861).
- Belford, P.C., Donahoo, J.D., & Heard, W.J. An evaluation of the effectiveness of software engineering techniques. In Proceedings of COMPCON '77. New York: IEEE, 1977.
- Black, R.K.E. Effects of modern programming practices on software development costs. In Proceedings of COMPCON '77. New York: IEEE, 1977.
- Boehm, B.W. Software and its impact: A quantitative assessment. Datamation, 1973, 19 (5), 48-59.
- Brown, J.R. Modern programming practices in large scale software development. In Proceedings of COMPCON '77. New York: IEEE, 1977.
- Caine, S.H., Farber, & Gordon, E.K. S-Fortran language reference guide. Pasadena, CA: Caine, Farber, & Gordon, Inc., 1974.
- DeRoze, B.C. Software research and development technology in the Department of Defense. Paper presented at the AIIE Conference on Software, Washington, D.C., December 1977.
- DeRoze, B.C., & Nyman, T.H. The software life cycle - A management and technological challenge in the Department of Defense. IEEE Transactions on Software Engineering, 1978, 4, 309-318.
- Dijkstra, E.W. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, & C.A.R. Hoare (Eds.), Structured Programming. New York; Academic Press, 1972.
- Duvall, L.M. The design of a software analysis center. In Proceedings of COMPSAC '77. New York: IEEE, 1977.
- Halstead, M.H. Natural laws controlling algorithm structure. Sigplan Notices, 1972, 7, (2), 19-26.

- Halstead, M.H. Elements of Software Science. New York; Elsevier North-Holland, 1977.
- Hecht, H., Sturm, W.A., & Trattner, S. Reliability measurement during software development (NASA-CR-145205). Hampton, VA: NASA Langley Research Center, 1977.
- Jones, T.C. Measuring programming quality and productivity. IBM Systems Journal, 1978, 17 (1), 39-63.
- Katzen, H. Systems Design and Documentation: An Introduction to the HIPO Method. New York: Van Nostrand Reinhold, 1976.
- Lyons, E.A., & Hall, R.R. ASTROS: Advanced Systematic Techniques for Reliable Operational Software. Lompac, CA: Vandenburg, AFB, Space and Missile Test Center, 1976.
- Maxwell, F.D. The determination of measures of software reliability (NASA-CR-158960). Hampton, VA: NASA Langley Research Center, 1978.
- McCabe, T.J. A complexity measure. IEEE Transactions on Software Engineering, 1976, 2, 308-320.
- McCall, J.A., Richards, P.K., & Walters, G.F. Factors in software quality (Tech. Rep. 77C1S02). Sunnyvale, CA: General Electric, Command and Information Systems, 1977.
- Mills, H.D. Mathematical foundations for structured programming. In V.R. Basili and T. Baker (Eds.), Structured Programming. New York: IEEE, 1975.
- Myers, G.J. Composite/Structured Design. New York: Van Nostrand Reinhold, 1978.
- Myers, W. The need for software engineering. Computer, 1978, 11 (2), 12-26.
- Nelson, R. Software data collection and analysis (Unpublished report). Rome, NY: Griffiss AFB, Rome Air Development Center, 1978.
- Putnam, L.H. A general empirical solution to the macro software sizing and estimating problem. IEEE Transactions on Software Engineering, 1978, 4, 345-361.

Salazar, J.A., & Hall, R.R. ASTROS Advanced Systematic Techniques for Reliable Operational Software: Another Look. Lompac, CA: Vandenburg, AFB, Space and Missile Test Center, 1977.

Sheppard, S.B., Curtis, B., Milliman, P., & Love, T. Human factors experiments on modern coding practices. Computer, 1979, 12, in press.

Stay, J.F. HIPO and integrated program design. IBM Systems Journal, 1976, 15, 143-154.

Stevens, W.P., Myers, G.J., & Constantine, L.L. Structured design. IBM Systems Journal, 1974, 13, 115-139.

Tausworthe, R.C. Standardized Development of Computer Software (2 vols.). Englewood Cliffs, NJ: Prentice-Hall, 1979.

Thayer, T.A., et al. Software reliability study (RADC-TR-76-238). Rome, NY: Griffiss AFB, Rome Air Development Center, 1976.

Walston, C.E., & Felix, C.P. A method of programming measurement and estimation. IBM Systems Journal, 1977, 18 (1), 54-73.

Yourdon, E., & Constantine, L.L. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. Englewood Cliffs, NJ: Prentice-Hall, 1979.

Yourdon Report (Vol. 1, No. 9). New York: Yourdon, Inc. 1976.